Project 1: Stacks and Queues

Due: 01/31/2024 at 11:59PM

General Guidelines:

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not need to be public should generally be kept private (instance variables, helper methods, etc.). Additional file creation isn't allowed as Vocareum will only compile the files given in the startercode.

In general, you are not allowed to import any additional classes or libraries in your code without explicit permission from the instructor! Adding any additional imports will result in a 0 for the entire project.

Note on Academic Dishonesty:

Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. **You MUST do your own work**! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online.

Note on implementation details:

Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

Note on grading and provided tests:

The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Project Overview:

In this project, you will work on two problems. First, we will implement a Double Ended Cyclic queue, and then we will use that queue to find a solution to a maze or generate a maze with a given seed.

Part 1: Double Ended Cyclic Queue (40 Points):

In part 1, you must implement a **Double Ended Cyclic Queue**. Adding an element is called enqueue and removing an element is called dequeue. In a traditional queue, elements are always enqueued at the tail of the list and dequeued from the head. However, in this version, we allow enqueue and dequeue at both the front and the back of the queue. We will implement a cyclic queue to utilize memory better. The underlying data structure that we will use is a dynamically resizable circular array.

Main idea:

Memory in computers is not circular, however, we can make it behave like a ring. The idea is to circle back when the end of the memory is reached. Another important factor is that Arrays have a fixed size; Queues do not. Therefore, we must occasionally resize the array. The guideline for resizing the array is as follows:

1. When the array becomes full, initialize a new array with

   *length = (old length) * increaseFactor*

   Copy the elements over from the old array to the new array. When copying, reorganize the array such that the front element is back at index 0 in the new array. I.e., the array is linear after resizing.

   Note: only resize when an enqueue would cause an overflow. If an enqueue causes the array to become full, do not prematurely resize. Wait until the next enqueue, and resize before inserting the new element.

2. If the capacity of the array is larger than 4 times the size of the array, reduce the size of the array by the *decreaseFactor*.

   length = (old length) / decreaseFactor

   However, the size should not be shrunk further than the initial capacity. Say for example, the initial capacity is 7, the array size should never be reduced below 7.

   Note: you should shrink the array directly after a dequeue or pop that causes the above situation. Do not wait until the next dequeue / pop.

You are encouraged to have a look at the API below and the starter code before reading the handout further. Please also have a look at the documentation comments in the starter code.

Class: cylic_double_queue:

In cylic_double_queue.hpp, you should implement a template class named cylic_double_queue. Since this is a template class, there's no .cpp file involved. The definition of each function/method of this class should be written inside the header file.

Public methods/function to implement:

1. You will find the functions and class members listed within the starter codes provided in this project for you to implement. Documentations of each function and class members are also provided. Please read carefully before you get started.
2. For the following functions:

    1. *T dequeue_front();*
    2. *T dequeue_back();*
    3. *void pop_front();*
    4. *void pop_back();*
    5. *T& front();*
    6. *T& back();*

    Make sure you throw *std::out_of_range("cyclic_double_queue is empty!")* if the queue is empty. Correctly throwing the required exceptions will be tested!

3. For the following function:
    1. *std::string print_status() const;*

    You should return a string that prints the occupancy of each slot in the array. Represent the slot with "[" and "]", and use "+" or "-" to represent whether the slot is occupied or not. The sample string output will be provided for each example below.

    Note: You should not add any kind of delimiter such as "\n" at the end of the string.

You must use an array as an underlying data structure. Using any other data structure will result in a zero for the entire project.

Example 1:

For the following examples, the queue stores characters (i.e., the template typename T is *char*). The queue below results after a series of operations:

Front = 5, Back = 6

Index	0	1	2	3	4	5	6
Contents	[]	[]	[]	[]	[]	[D]	[]

Output string from *print_status()*:

[-][-][-][-][-][+][-]

enqueue_back(A):

Front = 5, Back = 0

Index	0	1	2	3	4	5	6
Contents	[]	[]	[]	[]	[]	[D]	[A]

Output string from *print_status()*:

[-][-][-][-][-][+][+]

enqueue_back(F):

Front = 5, Back = 1

Index	0	1	2	3	4	5	6
Contents	[F]	[]	[]	[]	[]	[D]	[A]

Output string from *print_status()*:

[+][-][-][-][-][+][+]

enqueue_back(G):

Front = 5, Back = 2

Index	0	1	2	3	4	5	6
Contents	[F]	[G]	[]	[]	[]	[D]	[A]

Output string from *print_status()*:

[+][+][-][-][-][+][+]

enqueue_back(C):

Front = 5, Back = 3

Index	0	1	2	3	4	5	6
Contents	[F]	[G]	[C]	[]	[]	[D]	[A]

Output string from *print_status()*:

[+][+][+][-][-][+][+]

enqueue_front(W):

Front = 4, Back = 3

Index	0	1	2	3	4	5	6
Contents	[F]	[G]	[C]	[]	[W]	[D]	[A]

Output string from *print_status()*:

[+][+][+][-][+][+][+]

dequeue_front():

Front = 5, Back = 3

Index	0	1	2	3	4	5	6
Contents	[F]	[G]	[C]	[]	[]	[D]	[A]

Output string from *print_status()*:

[+][+][+][-][-][+][+]

dequeue_back():

Front = 5, Back = 2

Index	0	1	2	3	4	5	6
Contents	[F]	[G]	[]	[]	[]	[D]	[A]

Output string from *print_status()*:

[+][+][-][-][-][+][+]

Example 2:

Increasing the size when increase factor is 2:

(Queue in the example results after a series of operations)

Front = 3, Back = 2

Index	0	1	2	3	4	5	6
Contents	[E]	[F]	[]	[A]	[B]	[C]	[D]

Output string from *print_status()*:

[+][+][-][+][+][+][+]

enqueue_back(G):

Front = 3, Back = 3

Index	0	1	2	3	4	5	6
Contents	[E]	[F]	[G]	[A]	[B]	[C]	[D]

Output string from *print_status()*:

[+][+][+][+][+][+][+]

enqueue_back(H):

Front = 0, Back = 8

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Contents	[A]	[B]	[C]	[D]	[E]	[F]	[G]	[H]	[]	[]	[]	[]	[]	[]

Output string from *print_status()*:

[+][+][+][+][+][+][+][+][+][-][-][-][-][-][-][-]

Other Information:

1. We use a template class for this part of the project. For more information about generic classes please refer to:
    1. https://isocpp.org/wiki/faq/templates
    2. https://en.cppreference.com/w/cpp/language/class_template
2. The name "T" is a type parameter, a symbolic placeholder for some concrete type to be used.
3. The class will be instantiated as follows (using type *std::string*):
   auto queue = cyclic_double_queue<std::string>();
4. Therefore, implementing this template class will enable you to make a queue of integers, strings, and other complex data types as well.

Testing:

Correct implementation of cyclic_double_queue will give you 40 points.

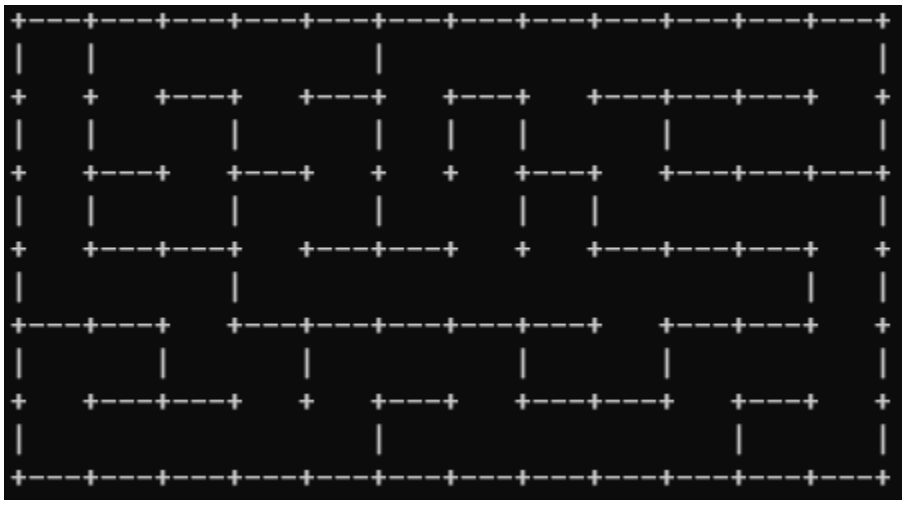
You are not allowed to include any extra headers. Including any headers, besides the ones in the starter code will give you a 0.

This part is REQUIRED for the implementation of part 2. The test cases will aggressively check for operation correctness in a randomized sequence. As such, if an earlier operation fails, later operations may also fail even if they are implemented correctly. You should ensure that your queue runs correctly before starting on part 2. See the section on Building and Testing at the end of the document for details.

Part 2a: Creating a Maze (30 points)

You are responsible for designing a 3D maze game. As you do not have access to the internet you can only use the *cyclic_double_queue* you implemented before entering the dungeon. Luckily the *cyclic_double_queue* can function both as a traditional stack and as a traditional queue. You will firstly initialize the maze map for the player given the dimensions and the random seed for the map, and then solve the maze by marking the path in the map.

Let's consider a 2D maze as a 2D array of spaces, where between each space, there is a wall or not. The starting point of the maze will be the upper left and the finish point is the lower right. A possible map with dimension (12, 6) could be:



'|', '---' represents the walls of each space and '+' represents the corners of each space.

For this assignment, you will work with 3D mazes instead of 2D mazes. Notice that the 3D maze has a width, height, and length, and we index spaces in the maze with (x, y, z).

Main idea:

The goal is to generate a maze from an array of 3D cells. Each cell in the maze is blocked by up to 6 adjacent walls, one for each possible direction. The value stored in the array at each cell location indicates which walls are currently blocking the cell (see below). At the start, all cells are blocked in all 6 directions. To generate the maze, the idea is to explore cells starting from a starting cell and moving in random directions, removing walls along the way, until every cell has been visited. By exploring in this way, you will ensure that there exists a path between any two cells through a series of reachable cells.

The algorithm to ensure all reachability will work as follows:

- *Initialize a stack with the start coordinate (0,0,0).*
- *Loop until the stack is empty:*
  - *Get the current coordinate at the top of the stack and mark it as visited.*
  - ***Retrieve a random direction as the test direction**.*
  - *Find the target neighbor coordinate in the test direction.*
  - *If the target neighbor cell is invalid or already visited:*
    - *Reassign a new test direction*
    - *Test again until all directions are tested.*
  - *If the target neighbor cell is valid and unvisited:*
    - *Remove the wall between the target neighbor cell and the current cell.*
    - *Push the target neighbor coordinate onto the stack.*
  - *If the current coordinate has no valid, unvisited neighbors:*
    - *Pop the current coordinate off the stack*

In this pseudocode, a valid neighbor is one that is within the dimensions of the maze. For example, in a maze of size 3x3x3, the cell at (1,2,0) is valid, but the cell at (1,3,0) is invalid since the Y value 3 is outside of the maze dimensions. You should consider all 6 directions to be potential neighbors to visit, but they might not all have valid coordinates.

Looking over this routine, you should be able to envision how this procedure will dive through the maze randomly, like a snake, until the snake reaches a dead end (a space without any unvisited neighbors). The exploration of the snake would then have to backtrack until there is a space with unvisited neighbors and explore from there. Eventually, the snake will have explored the entire maze, at which point, you're done.

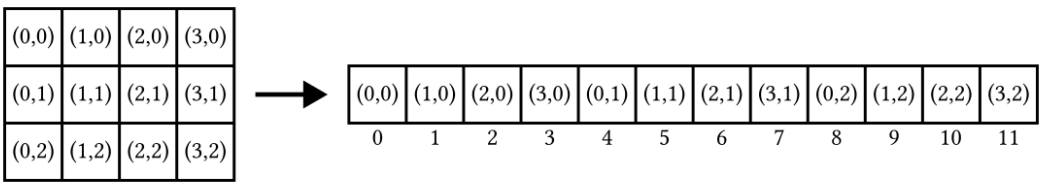
The left bottom front index is (0, 0, 0), and the right top back index is (width - 1, height - 1, length - 1). In other words, the Left direction corresponds to -X, Bottom is -Y, Front is -Z, and vice versa.

Maze representation:

The maze should be represented as a 1-dimensional array of *unsigned* integer values. Despite the fact that this maze is 3D, the array is 1D. You may have to convert between array indices and 3D coordinates during maze generation (e.g., when determining neighbor indices). A general formula is as follows:

index = X * x_stride + Y * y_stride + Z * z_stride

The stride is the number of array indices between consecutive coordinate values. For example, given the following 2D grid on the left, we can represent it with the 1D array on the right like so:



Starting from coordinate (1,1) at index 5 in the array, we move one cell in the +X direction to arrive at (2,1) at index 6 in the array. Consecutive X coordinates are stored consecutively in the array, so the x_stride is 1. From (2,1) at index 6, we move one cell in the +Y direction to arrive at (2,2) at index 10. In this case, consecutive Y coordinates are separated by *width* array indices, so the y_stride is *width*. Extend this pattern to 3D when transforming between 3D maze coordinates and array indices. You can derive a similar expression to determine the X, Y, and Z coordinates given the array index.

Directions and Walls:

In the starter code, an enum class named "Direction" is provided. It contains a sequence of directions (Right, Left, Top, Bottom, Back, Front) and a special member (Mark) with preset values for bit operations. Each *unsigned* value in the maze array is a combination of these Directions that represents whether there are any walls in all 6 directions, and whether this space is marked or not. The Directions have the following values:

Direction::Right	= (1 << 0)	= 00000001	= 1
Direction::Left	= (1 << 1)	= 00000010	= 2
Direction::Top	= (1 << 2)	= 00000100	= 4
Direction::Bottom	= (1 << 3)	= 00001000	= 8
Direction::Back	= (1 << 4)	= 00010000	= 16
Direction::Front	= (1 << 5)	= 00100000	= 32
Direction::Mark	= (1 << 6)	= 01000000	= 64

For example, the *unsigned* value of a space which is marked as part of the path, and contains walls on right, bottom, back, and front will be:

(*Direction::Right) + (Direction::Bottom) + (Direction::Back) + (Direction::Front) + (Direction::Mark)*
= (1 << 0) + (1 << 3) + (1 << 4) + (1 << 5) + (1 << 6) = 01111001
= 1 + 8 + 16 + 32 + 64 = 121

The *unsigned* value of a space which is not marked and contains walls on left, top, and back will be:

(*Direction::Left) + (Direction::Top) + (Direction::Back)*
= (1 << 1) + (1 << 2) + (1 << 4) = 00010110
= 2 + 4 + 16 = 22

You should use bitwise operations to add, remove, or detect walls, and to mark or unmark spaces.

Deterministic random direction generator and reassigning new directions.

IMPORTANT: Please carefully read this part! If you use the random direction generation function incorrectly, you will receive a zero for all tests of creating the maze!

A simple deterministic random direction generator is provided in the starter code. Given the same seed value, it will always produce the same sequence of random directions. **The test cases will test your result by giving a random seed for creating the maze. So, you should never try to record or memorize the seed or desired output from the test cases.** The API for retrieving a new random direction is:

static Direction get_next_direction(unsigned& seed);

The part of pseudo code in bold, and highlighted with blue color is the part of the code you should call the API. The part of the pseudo code underlined, and highlighted with green color is the part of the code you should reassign according to the order, instead of calling to the API. Each time you pop a new coordinate from the stack, you call this API ONLY ONCE to retrieve the first test direction. The generator will change the value of seed and return the new direction for you.

From the pseudo code provided above, if the target coordinate of the test direction is visited or it's out of the boundary of the map, you should directly reassign a new direction following the order below:

…Front->Right->Left->Top->Bottom->Back->Front->Right…

The order of directions is the same as declared in the *Direction* enum class.

Printing walls of the maze:

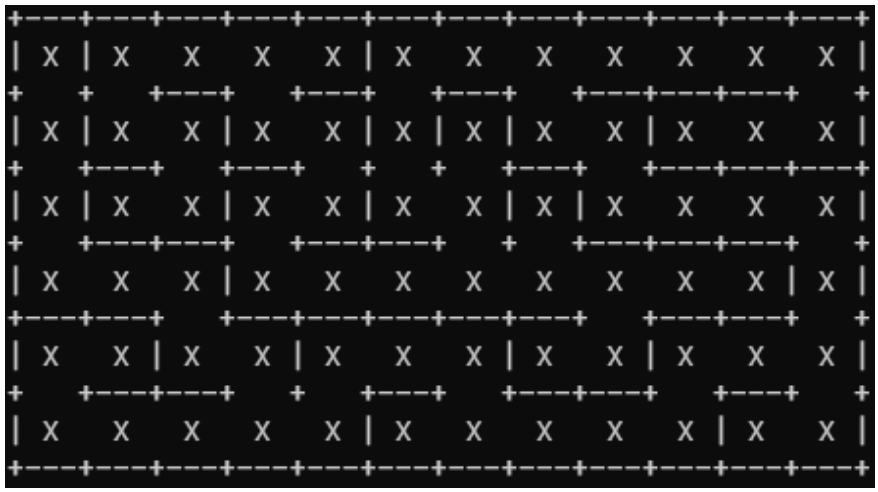
In the starter code, you need to implement:

std::string print_walls(size_t z = 0) const;

to get a string representation of the walls of the maze. You can use the std::stringstream class to build the string and return it. Since we are working on a 3D maze, different from the 2D example output above, we add 'B', 'F', 'X', and <*space*> to represent the status of walls on the front and the back direction of the cell.

The 'B' represents the current cell containing walls on the back, but not the front.
The 'F' represents the current cell containing walls on the front, but not the back.
The 'X' represents both the front, and the back directions being blocked by walls.
The <*space*> represents that there's no wall in either the front or the back direction.

With the newly added characters, the map of the previous example (12x6x1) now should be:



For a 3x3x3 maze, if we print all 3 layers by executing print_walls(0), print_walls(1), print_walls(2), a possible output could be:

Z = 0 Z = 1 Z = 2

Important Note: although the coordinate (0,0,0) is described as the “left, bottom, front” corner of the maze, when printing layer Z, the coordinate (0,0,Z) should appear in the **top** left corner of the output. In other words, the first row of cells in the printout is actually the “bottom” row of cells, and the last row of cells is the “top”.

Public methods/function to implement:

1. You will find the functions listed within the starter code provided in this project for you to implement. Documentations of each function are also provided. Please read carefully before you get started.
2. You should implement the following functions for this part:
    1. *void initialize(size_t dimX, size_t dimY, size_t dimZ, unsigned seed);*
    2. *std::string print_walls(size_t z = 0) const;*
    3. *void get_dim(size_t& dimX, size_t& dimY, size_t& dimZ) const;*
    4. *void get_index(size_t x, size_t y, size_t z) const;*

Part 2b: Solving a Maze (30 points)

In this part, you will use your generated maze from the previous part to find paths between start points and end points within the maze.

The process of solving a maze with a queue is a lot like "hunting" out the end point from the start point. The queue will allow us to track which spaces we should visit next. In addition, we need to keep a record of how we arrived at each space, so that when we’ve found the endpoint we can trace our steps back to the start and get the final path.

A general searching procedure is as follows:

- *Initialize a queue with the start coordinate.*
- *Initialize a "previous" array of the same size as the map*
- *Loop until the queue is empty:*
    - *Dequeue current index and mark it as visited.*
    - *If current index is the end point:*
        - *Break! We've found the end.*
    - *For each valid, unvisited neighbor of the current index:*
        - *Enqueue the neighbor's index*
        - *Set the neighbor's "previous" index to the current index*
- *If the end point was found:*
    - *Set the current index to the end point*
    - *Loop until the current index is the start point:*
        - ***Mark** the current index as part of the path using Direction::Mark*
        - *Set the current index to its "previous" index*
    - ***Mark** the start index as part of the path using Direction::Mark*

Note that the start coordinate and end coordinate are both given as arguments to the *solve()* function, so it will not always begin at (0,0,0) and end at (dimX-1, dimY-1, dimZ-1).

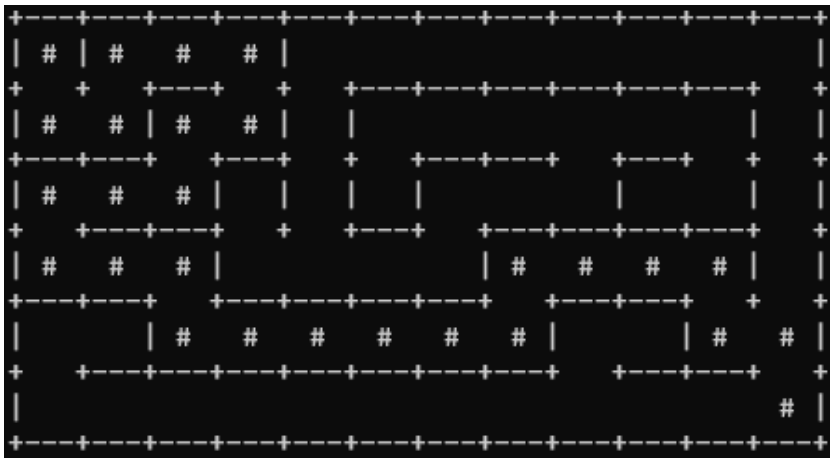
Printing marks of the maze:

In the starter code, you need to implement:

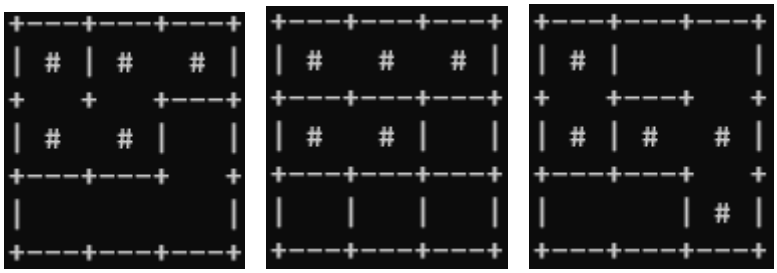
std::string print_marks(size_t z = 0) const;

Similar to *std::string print_walls(size_t z = 0) const*, you need to print a specific layer of the maze. Here we replace the front/back wall with Marks. If a cell is marked, print the '#' character, otherwise, print a <*space*>. When this method is called after calling *solve()* above, we should see the path from the starting coordinate to the ending coordinate.

For example, if we want to print the path of a 12x6x1 maze, the output could look like:



If we want to print the marked path for a 3x3x3 maze by executing print_marks(0), print_ marks(1), print_ marks(2), the output might be:



Z = 0 Z = 1 Z = 2

Public methods/function to implement:

1. You will find the functions listed within the starter codes provided in this project for you to implement. Documentations of each function are also provided. Please read carefully before you get started.
2. You should implement the following functions for this part:
    1. *bool solve(size_t startX, size_t startY, size_t startZ, size_t endX, size_t endY, size_t endZ);*
    2. *std::string print_marks(size_t z = 0) const;*

Building and Testing Your Code:

Included with the starter code is a CMake configuration file. CMake is a cross-platform build system that generates project files based on your operating system and compiler. The general procedure for building your program is to run CMake, and then build using the generated project files or makefile. On Linux that process looks like this:

1. Navigate to your project directory, where CMakeLists.txt is located
2. Create a build directory for compiled executables:
   `$ mkdir build`
   Note: the $ symbol indicates the shell prompt - do not actually type $
3. Navigate to the new build directory
   `$ cd build`
4. Run CMake, which generates a Makefile
   `$ cmake ..`
   The .. indicates the parent directory where your CMake configuration is
5. Compile the code using the Makefile
   `$ make`

For Windows the process is similar, except CMake will typically generate a Visual Studio solution. If you want to compile with debug symbols, replace step 4 with

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

If successful, the program will compile two executables: queue-app and maze-app. These programs accept input from stdin and print to stdout. You can use them to test your queue and maze implementations. For details on how they work, consult the source code.

Provided with the starter code are several sample inputs and expected outputs for queue-app and maze-app. To see if your implementation is correct, run the programs with the inputs, and compare the output with the expected outputs. For example:

```
$ ./queue-app < ../sample_tests/input/queue_test_00.txt > queue_test_00_output.txt
$ diff -qsZB queue_test_00_output.txt ../sample_tests/expected/queue_test_00.txt
```

If diff reports the files as identical, then your implementation matches the expected output.

You should also test your code on Vocareum. After uploading your files, press the **Run** button on the top right. This will execute a script that will compile your code and run it against all of the sample test cases, and print a summary of which tests failed. You should run your code before submitting it to make sure it compiles and executes properly, since you will have a limited number of submissions.

Note that the sample test cases are not exhaustive and purposefully do not cover all possible cases. When you submit your code it will be tested against many more tests with more thorough cases, which will not be made available to inspect. It is your responsibility to ensure your program is correct, by devising new tests and manually inspecting their outputs.